On Linear Logic, Functional Programming, and Attack Trees

Harley Eades III[1], Jiaming Jiang[2], and Aubrey Bryant[3]

[1] Computer Science, Augusta University, harley.eades@gmail.com
[2] Computer Science, North Carolina State University
[3] Computer Science, Augusta University

Abstract. This paper has two main contributions. The first is a new linear logical semantics of causal attack trees in four-valued truth tables. Our semantics is very simple and expressive, supporting specializations, and supports the *ideal* semantics of causal attack trees, and partially supporting the *filter* semantics of causal attack trees. Our second contribution is Lina, a new embedded, in Haskell, domain specific functional programming language for conducting threat analysis using attack trees. Lina has many benefits over existing tools; for example, Lina allows one to specify attack trees very abstractly, which provides the ability to develop libraries of attack trees, furthermore, Lina is compositional, allowing one to break down complex attack trees into smaller ones that can be reasoned about and analyzed incrementally. Furthermore, Lina supports automatically proving properties of attack trees, such as equivalences and specializations, using Maude and the semantics introduced in this paper.

1 Introduction

Attack trees are perhaps the most popular graphical model used to conduct threat analysis of both physical and virtual secure systems. They were made popular by Bruce Schneier in the late nineties [16]. In those early years attack trees were studied and used as a syntactic tool to help guide analysis. However, as systems grew more complex the need for a semantics of attack trees become apparent; after all, without a proper semantics how can we safely manipulate attack trees, extend their expressivity, or compare them?

A number of different models of attack trees have been proposed: a model in boolean algebras [11,10,15], series-parallel pomsets [12], Petri nets [13], and tree automata [1]. There have also been various extensions, such as, adding sequential composition [6], and defense nodes [9,10]. All of these models and extensions have their benefits, but at the heart of them all is logic.

The model in boolean algebras was the first and most elegant model of attack trees, but it failed to capture the process aspect of attack trees, that is, the fact that base attacks are actual processes that need to be carried out, and the branching nodes compose these processes in different ways. Thus, the community moved towards models of resources like parallel-series pomsets, Petri nets,

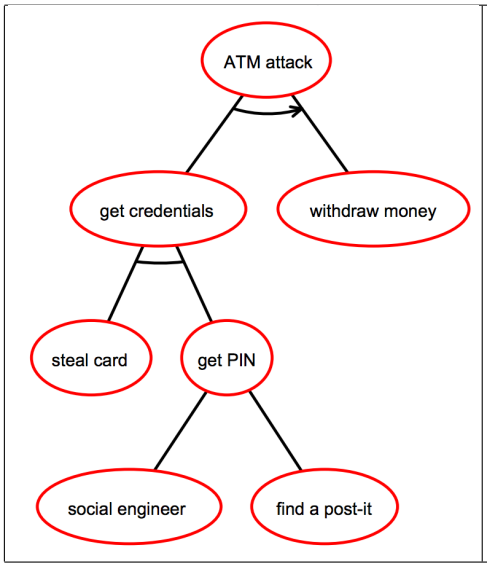
and automata. However, the complexity of these models increased, and hence, comparing these models becomes difficult. Furthermore, this increased complexity makes it hard to decide which to use and under which circumstances. This difficulty can be resolved by recovering the elegant logical model of attack trees.

Linear Logic. It is fitting that attack trees are the most popular model used in threat analysis, because *linear logic*, one of the most widely studied logics used to reason about resources, is also an excellent candidate for modeling attack trees. In fact, Horne et al.[5] has already produced a number of interesting results. Most importantly, they show that attack trees can be modeled as formulas in linear logic, which then one can prove properties between attack trees by proving implications between them. Furthermore, by studying attack trees from a linear logical perspective they introduce a new property between attack trees called *specializations*. Prior to their paper the literature was primarily concerned with equality between attack trees, but the logical semantics of attack trees reveal how one can break these equalities up into directional rewrite rules. An attack tree is a *specialization* of another if the former is related to the later via these rewrite rules. The logical semantics model the rewrite rules as implications.

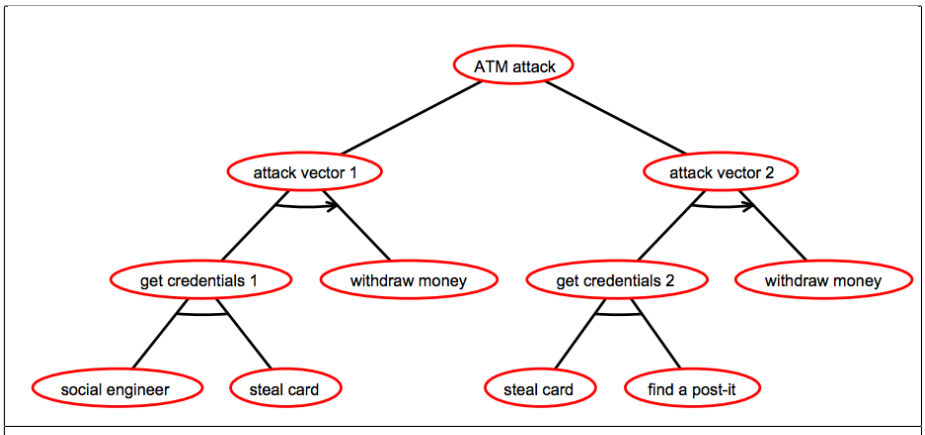
This paper has two main contributions. The first is a new simple linear logical semantics of causal attack trees – attack trees with sequential composition – in four-valued truth tables. It comes in two flavors: the ideal quaternary logic (Section 3.1) and the filterish quaternary logic (Section 3.2). These two types of semantics correspond to truth table semantics for Horne et al.'s[5] *ideal* and *filter* semantics of causal attack trees.

Functional Programming. Our second contribution is Lina, a new domain specific functional programming language for conducting threat analysis using attack trees. Consider the example attack trees in Fig. 1. Both of these contain actual Lina programs for each of the corresponding attack trees; in fact, every example in this paper is a Lina program. Lina supports causal attack trees with attributes or without; thus, there are two types of base attacks: base attacks with attributes, denoted `base_wa`, and base attacks with no attributes, denoted `base_na`; an example usage of the former can be found in Fig. 3. Lina is designed to be extremely simple, and to reflect the typical pseudocode found throughout the literature. However, Lina is more than just a simple definitional language.

Lina is an embedded domain-specific programming language whose host language is the Haskell programming language [7]. So, why Haskell? As security researchers and professionals, we are in the business of verifying the correctness of various systems. Thus, we should be taking advantage of verification tools to insure that our constructions, tools, and analysis are correct. By embedding Lina into Haskell, we are able to take advantage of cutting-edge verification tools while conducting threat analysis. For example, right out the box Lina supports property-based randomized testing using QuickCheck [2], and refinement types in Liquid Haskell [17] to verify properties of our attack trees or the attribute domains used while analyzing attack trees. Furthermore, Haskell's advanced type system helps catch bugs while we develop our attack trees and their attribute




```
seq_node "ATM attack"
  (and_node "get credientials"
    (base_na "steal card")
    (or_node "get PIN"
      (base_na "social engineer")
      (base_na "find a post-it")))
  (base_na "withdraw money")
```




```
or_node "ATM attack"
   (seq_node "attack vector 1"
     (and_node "get credentials 1"
       (base_na "social engineer")
       (base_na "steal card"))
     (base_na "withdraw money"))
   (seq_node "attack vector 2"
     (and_node "get credentials 2"
       (base_na "steal card")
       (base_na "find a post-it"))
     (base_na "withdraw money"))
```

Fig. 1. Attack Trees for an ATM attack from Figure 1 and Figure 2 of Kordy et al. [8] and their corresponding Lina scripts.

domains as a side-effect of type checking. Finally, functional programs are short, but not obfuscated, and hence allow for very compact and trustworthy programs.

That being said, we are designing Lina so that it can be used with very little Haskell experience. It is our hope that one will be able to make use of Lina without knowing Haskell, and we plan to develop new tooling to support this.

Lina approaches threat analysis from a programming language perspective, leading to a number of new advances. First, as Gadyatskaya and Trujillo-Rasua [4] argue, as a community we need to start building more automated means of conducting threat analysis, and there is no better way to build or connect automated tools than a programming language. Lina is perfect as a target for new tools, and it can be connected to existing tools fairly easily. In fact, Lina already supports automation using the automatic rewrite system Maude [3]; for example, the two attack trees in Fig. 1 can be automatically proven equivalent to each other in Lina. This is similar to Krody's [8] SPTool, but Lina goes further and supports more than one backend rewrite system; for example, Lina is the first tool to support automatically proving specializations of attack trees. The user can choose which backend they wish to use.

2 Causal Attack Trees

We begin by introducing causal attack trees. This formulation of attack trees was first proposed by Jhawar et al. [6], where they called them SAND attack trees, however sequential composition does not always maintain the same properties as conjunction; for example, classically it is a self dual operator. Thus, we follow Horne et al.'s lead [5] and call them causal attack trees.

Definition 1. *Suppose $\mathbb{B}$ is a set of base attacks whose elements are denoted by b. Then an **attack tree** is defined by the following grammar:*

$$A, B, C, T := b \mid \mathsf{OR}(A, B) \mid \mathsf{AND}(A, B) \mid \mathsf{SEQ}(A, B)$$

Equivalence of attack trees, denoted by $A \approx B$, is defined as follows:

$\mathsf{OR}(A, A) \approx A$	$\mathsf{OR}(\mathsf{OR}(A, B), C) \approx \mathsf{OR}(A, \mathsf{OR}(B, C))$
$\mathsf{OR}(A, B) \approx \mathsf{OR}(B, A)$	$\mathsf{AND}(\mathsf{AND}(A, B), C) \approx \mathsf{AND}(A, \mathsf{AND}(B, C))$
$\mathsf{AND}(A, B) \approx \mathsf{AND}(B, A)$	$\mathsf{SEQ}(\mathsf{SEQ}(A, B), C) \approx \mathsf{SEQ}(A, \mathsf{SEQ}(B, C))$
	$\mathsf{AND}(A, \mathsf{OR}(B, C)) \approx \mathsf{OR}(\mathsf{AND}(A, B), \mathsf{AND}(A, C))$
	$\mathsf{SEQ}(A, \mathsf{OR}(B, C)) \approx \mathsf{OR}(\mathsf{SEQ}(A, B), \mathsf{SEQ}(A, C))$

Throughout the sequel we will show that the previous rules are sound with respect to our new model, but just as Horne et al. [5] did, we will then show that there are properties of attack trees that these rules do not support, but our semantics allows, for example, the rules given in Lemma **??** cannot be modeled using these rules.

3 A Quaternary Semantics for Causal Attack Trees

Kordy et al. [10] gave a very elegant and simple semantics of attack-defense trees in boolean algebras. Unfortunately, while their semantics is elegant, it does not

capture the resource aspect of attack trees, it allows contraction, and it does not provide a means to model sequential composition. In this section we give a semantics of attack trees in the spirit of Kordy et al.'s using a four-valued logic. This section was formally verified in the Agda Proof Assistant [14][4].

We now give two types of quaternary semantics for casual attack trees. We do this by defining two four-valued logics we call quaternary logics. The propositional variables, elements of the set PVar, of our quaternary logics, denoted by P, Q, R, and S, range over the set $\mathsf{4} = \{0, \frac{1}{4}, \frac{1}{2}, 1\}$. We think of 0 and 1 as we usually do in boolean algebras, but we think of $\frac{1}{4}$ and $\frac{1}{2}$ as intermediate values that can be used to break various structural rules[5]. In particular we will use these values to prevent exchange for sequential composition from holding, and contraction from holding for parallel and sequential composition.

We use the usual notion of equivalence between propositions; that is, propositions ϕ and ψ are considered equivalent, denoted by $\phi \equiv \psi$, if and only if they have the same truth tables. In addition, we define a notion of entailment for the quaternary logics. Denote by $P \leq_4 Q$ the usual natural number ordering restricted to $\mathsf{4}$. Then we have the following result immediately.

Lemma 1 (Entailment in the Quaternary Logics). *$P \equiv Q$ if and only if $P \leq_4 Q$ and $Q \leq_4 P$*

This result shows that we can break up the equivalence of attack trees into directional properties captured here by entailments, and hence, every equivalence proved throughout this section can also be used directionally.

3.1 The Ideal Quaternary Logic

The ideal semantics for casual attack trees was first proposed by Horne et al.[5]. In this section we give a simple truth table semantics that corresponds to their ideal semantics within the ideal quaternary logic.

Definition 2. *The logical connectives of the* ideal quaternary logic *are defined as follows:*

Parallel Composition:
$P \odot_I Q = 1$,
  where neither P nor Q are 0
$P \odot_I Q = 0$, *otherwise*

Sequential Composition:
$P \rhd_I Q = \frac{1}{2}$,
  where $P \in \{\frac{1}{2}, 1\}$ and $Q \neq 0$
$P \rhd_I Q = \frac{1}{4}$,
  where $P = \frac{1}{4}$ and $Q \neq 0$
$P \rhd_I Q = 0$, *otherwise*

Choice:
$P \sqcup_I Q = \mathsf{max}(P, Q)$

[4] The formalization can be found at `https://github.com/MonoidalAttackTrees/ATLL-Formalization`

[5] Choosing $\frac{1}{4}$ and $\frac{1}{2}$ as the symbols for the intermediate values was arbitrary, and one can choose any symbols at all for these two values and the semantics will still be correct.

These definitions are carefully crafted to satisfy the necessary properties to model attack trees on the ideal semantics. Comparing these definitions with Kordy et al.'s [10] work we can see that choice is defined similarly, but parallel composition is not a product – ordinary conjunction – but rather a linear tensor product. Sequential composition is not actually definable in a boolean algebra, and hence makes use of the intermediate values to insure that neither exchange nor contraction hold.

In order to model attack trees, the previously defined logical connectives must satisfy the appropriate equivalences corresponding to the equations between attack trees. We break these properties up into the following lemmata.

Lemma 2 (Basic Properties for Choice). *The following properties hold:*

1. $(P \sqcup_I Q) \equiv (Q \sqcup_I P)$
2. $((P \sqcup_I Q) \sqcup_I R) \equiv (P \sqcup_I (Q \sqcup_I R))$
3. $P \leq_4 (P \sqcup_I Q)$
4. $Q \leq_4 (P \sqcup_I Q)$
5. *If* $P \leq_4 R$ *and* $Q \leq_4 R$, *then* $(P \sqcup_I Q) \leq_4 R$
6. *If* $P \leq_4 R$ *and* $Q \leq_4 S$, *then* $(P \sqcup_I Q) \leq_4 (R \sqcup_I S)$

Proof. Each of the properties hold by comparing truth tables.

The previous lemma shows that choice has the same properties as boolean disjunction. Hence, it is possible to show using these rules that $P \sqcup_I P \equiv P$ which follows from properties three, four, and five.

Lemma 3 (Basic Properties for Parallel Composition). *The following properties hold:*

1. $(P \odot_I P) \not\equiv P$
2. $(P \odot_I Q) \equiv (Q \odot_I P)$
3. $((P \odot_I Q) \odot_I R) \equiv (P \odot_I (Q \odot_I R))$
4. $(P \odot_I (Q \sqcup_I R)) \equiv ((P \odot_I Q) \sqcup_I (P \odot_I R))$
5. *If* $P \leq_4 R$ *and* $Q \leq_4 S$, *then* $(P \odot_I Q) \leq_4 (R \odot_I S)$

Proof. We give the proof of property one. The other properties hold by comparing truth tables. Suppose $P = \frac{1}{2}$, then $P \odot_I P = \frac{1}{2} \odot_I \frac{1}{2} = 1$, but 1 is not $\frac{1}{2}$.

The previous lemma shows that sequential composition is a linear tensor product. In particular, the first property guarantees that sequential composition does not contract parallel copies of attack trees into a single attack tree.

Lemma 4 (Basic Properties for Sequential Composition). *The following properties hold:*

1. $(P \rhd_I P) \not\equiv P$
2. $(P \rhd_I Q) \not\equiv (Q \rhd_I P)$
3. $(P \rhd_I (Q \rhd_I R)) \equiv ((P \rhd_I Q) \rhd_I R)$
4. $(P \rhd_I (Q \sqcup_I R)) \equiv ((P \rhd_I Q) \sqcup_I (P \rhd_I R))$
5. *If* $P \leq_4 R$ *and* $Q \leq_4 S$, *then* $(P \rhd_I Q) \leq_4 (R \rhd_I S)$

Proof. We give proofs for properties one and two, but the others hold by comparing truth tables. As for property one, suppose $P = 1$, then $P \rhd_I P = 1 \rhd_I 1 = \frac{1}{2}$, but 1 is not $\frac{1}{2}$. Now for property two, suppose $P = 1$ and $Q = \frac{1}{4}$, then $P \rhd_I Q = 1 \rhd_I \frac{1}{4} = \frac{1}{2}$, but $Q \rhd_I P = \frac{1}{4} \rhd_I 1 = \frac{1}{4}$.

This lemma is similar to the previous. However, property two guarantees that sequential composition is not commutative.

Lemma 5 (The Ideal Properties). *The following properties hold:*

1. $((P \sqcup_I Q) \rhd_I (R \sqcup_I S)) \leq_4 ((P \rhd_I R) \sqcup_I (Q \rhd_I S))$
2. $((P \sqcup_I Q) \rhd_I R) \leq_4 (P \sqcup_I (Q \rhd_I R))$
3. $(P \rhd_I (Q \sqcup_I R) \leq_4 (Q \sqcup_I (P \rhd_I R))$
4. $(P \rhd_I Q) \leq_4 (P \sqcup_I Q)$

Proof. Each property holds by comparing truth tables.

At this point it is quite easy to model attack trees as formulas. The following defines their interpretation.

Definition 3. *Suppose* $\mathbb{B}$ *is some set of base attacks, and* $\nu : \mathbb{B} \to \mathsf{PVar}$ *is an assignment of base attacks to propositional variables. Then we define the interpretation of attack trees to propositions as follows:*

$$\begin{array}{lcllcl} [\![b \in \mathbb{B}]\!] & = & \nu(b) & [\![\mathsf{SEQ}(A,B)]\!] & = & [\![A]\!] \rhd_I [\![B]\!] \\ [\![\mathsf{AND}(A,B)]\!] & = & [\![A]\!] \odot_I [\![B]\!] & [\![\mathsf{OR}(A,B)]\!] & = & [\![A]\!] \sqcup_I [\![B]\!] \end{array}$$

We can use this semantics to prove equivalences between attack trees.

Lemma 6 (Equivalence of Attack Trees in the Ideal Quaternary Semantics). *Suppose* $\mathbb{B}$ *is some set of base attacks, and* $\nu : \mathbb{B} \to \mathsf{PVar}$ *is an assignment of base attacks to propositional variables. Then for any attack trees* A *and* B, *if* $A \approx B$, *then* $[\![A]\!] \equiv [\![B]\!]$.

Proof. This proof holds by induction on the form of $A \approx B$.

3.2 The Filterish Quaternary Logic

We now introduce the filterish semantics for casual attack trees. This is a restricted notion of the filter semantics of Horne et al. [5]. We were unable to find a quaternary semantics for the full filter semantics, because we obtained contractions when attempting to satisfy the corresponding specialization properties in the filter model. We are unsure if these contradictions arise due to the fact that the semantics proposed here is intuitionistic while Horne et al. [5] use classical logic, or if four values just are not enough, or if we just have not been able to find it.

In this section we do as we did in the previous and define a quaternary logic called the *filterish quaternary logic*.

Definition 4. *The logical connectives of the* filterish quaternary logic *are defined as follows:*

Parallel Composition:
$P \odot_F Q = \frac{1}{2}$,
  where neither P *nor* Q *are* 0
$P \odot_F Q = 0$, *otherwise*

Sequential Composition:
$P \rhd_F Q = 1$,
  where $P \in \{\frac{1}{2}, 1\}$ *and* $Q \neq 0$
$P \rhd_F Q = \frac{1}{4}$,
  where $P = \frac{1}{4}$ *and* $Q \neq 0$
$P \rhd_F Q = 0$, *otherwise*

Choice:
$P \sqcup_F Q = \mathsf{max}(P, Q)$

We have the same basic properties as the ideal quaternary logic. We omit proofs, because they are similar to the corresponding properties in the ideal semantics.

Lemma 7 (Basic Properties for Choice). *The following properties hold:*

1. $(P \sqcup_F Q) \equiv (Q \sqcup_F P)$
2. $((P \sqcup_F Q) \sqcup_F R) \equiv (P \sqcup_F (Q \sqcup_F R))$
3. $P \leq_4 (P \sqcup_F Q)$
4. $Q \leq_4 (P \sqcup_F Q)$
5. *If* $P \leq_4 R$ *and* $Q \leq_4 R$, *then* $(P \sqcup_F Q) \leq_4 R$
6. *If* $P \leq_4 R$ *and* $Q \leq_4 S$, *then* $(P \sqcup_F Q) \leq_4 (R \sqcup_F S)$

Lemma 8 (Basic Properties for Parallel Composition). *The following properties hold:*

1. $(P \odot_F P) \not\equiv P$
2. $(P \odot_F Q) \equiv (Q \odot_F P)$
3. $((P \odot_F Q) \odot_F R) \equiv (P \odot_F (Q \odot_F R))$
4. $(P \odot_F (Q \sqcup_F R)) \equiv ((P \odot_F Q) \sqcup_F (P \odot_F R))$

5. *If* $P \leq_4 R$ *and* $Q \leq_4 S$*, then* $(P \odot_F Q) \leq_4 (R \odot_F S)$

Lemma 9 (Basic Properties for Sequential Composition). *The following properties hold:*

1. $(P \rhd_F P) \not\equiv P$
2. $(P \rhd_F Q) \not\equiv (Q \rhd_F P)$
3. $(P \rhd_F (Q \rhd_F R)) \equiv ((P \rhd_F Q) \rhd_F R)$
4. $(P \rhd_F (Q \sqcup_F R)) \equiv ((P \rhd_F Q) \sqcup_F (P \rhd_F R))$
5. *If* $P \leq_4 R$ *and* $Q \leq_4 S$*, then* $(P \rhd_F Q) \leq_4 (R \rhd_F S)$

We now give the filterish properties that correspond to a subset of the filter properties proposed by Horne et al. [5].

Lemma 10 (The Filterish Properties). *The following properties hold:*

1. $((P \rhd_F R) \odot_F (Q \rhd_F S)) \leq_4 ((P \odot_F Q) \rhd_F (R \sqcup_F S))$
2. $(P \sqcup_F (Q \rhd_F R)) \leq_4 ((P \sqcup_F Q) \rhd_F R)$

The remaining filter properties proposed by Horne et al. [5] actually fail in both directions.

Lemma 11. *There exists an* P*,* Q*, and* R *that cause the following properties to not hold:*

1. $(P \rhd_F (Q \odot_F R)) \leq_r (Q \sqcup_F (P \rhd_F R))$
2. $(P \rhd_F Q) \leq_4 (P \sqcup_F Q)$

Interestingly, if we change Definition 4 so that all the basic properties hold and Lemma 11 holds, then the inequalities in Lemma 10 degenerate to equalities. We were unable to find a definition of the logical connectives that make all of the properties in both of the previous lemmas hold.

Just as we did for the ideal quaternary semantics we can show that we can model attack trees as formulas. The following defines their interpretation.

Definition 5. *Suppose* $\mathbb{B}$ *is some set of base attacks, and* $\nu : \mathbb{B} \to \mathsf{PVar}$ *is an assignment of base attacks to propositional variables. Then we define the interpretation of attack trees to propositions as follows:*

$$\begin{array}{lcllcl} [\![b \in \mathbb{B}]\!] & = & \nu(b) & [\![\mathsf{SEQ}(A,B)]\!] & = & [\![A]\!] \rhd_F [\![B]\!] \\ [\![\mathsf{AND}(A,B)]\!] & = & [\![A]\!] \odot_F [\![B]\!] & [\![\mathsf{OR}(A,B)]\!] & = & [\![A]\!] \sqcup_F [\![B]\!] \end{array}$$

We can use this semantics to prove equivalences between attack trees.

Lemma 12 (Equivalence of Attack Trees in the Ideal Quaternary Semantics). *Suppose* $\mathbb{B}$ *is some set of base attacks, and* $\nu : \mathbb{B} \to \mathsf{PVar}$ *is an assignment of base attacks to propositional variables. Then for any attack trees* A *and* B*, if* $A \approx B$*, then* $[\![A]\!] \equiv [\![B]\!]$*.*

Proof. This proof holds by induction on the form of $A \approx B$.

3.3 An Example Specialization

The quaternary logics introduced in the previous section do indeed capture all of the equivalences of attack trees, but they also support proving specializations. Consider the example attack trees in Fig. 2. In the ideal semantics attack tree C is

A.

```
and_node "obtain secret"
 (or_node "obtain encryped file"
    (base_na "bribe sysadmin")
    (base_na "steal backup"))
 (seq_node "obtain password"
    (base_na "break into system")
    (base_na "install keylogger"))
```

B.

```
seq_node "break in, obtain secret"
 (base_na "break into system")
 (and_node "obtain secret inside"
   (base_na "install keylogger")
   (base_na "steal backup"))
```

C.

```
or_node "obtain secret"
 (and_node "obtain secret via sysadmin"
  (base_na "bribe sysadmin")
  (seq_node "obtain password"
    (base_na "break into system")
    (base_na "install keylogger")))
  (seq_node "break in, obtain secrect"
    (base_na "break into system")
    (and_node "obtain secret inside"
      (base_na "install keylogger")
      (base_na "steal backup")))
```

Fig. 2. Encrypted Data Attack from Figure 1 (A), Figure 3 (B), and Figure 2 (C) of Horne et al. [5].

a sound specialization of attack tree A, and attack tree B is a sound specialization of attack tree A. Attack tree C requires the attacker to break into the system before they can steal the backup, but attack tree A does not require this. Then attack tree B has dropped bribing the sysadmin and simply requires the attacker to just steal the backups. Notice that none of the attack trees in Fig. 2 are equivalent. So how do we prove these specializations are sound? We prove that they are related through an entailment rather than an equivalence.

Definition 6. *An attack tree A is a sound specialization of an attack B if and only if $[\![A]\!] \leq_4 [\![B]\!]$.*

We can now formally prove that the attack tree C is a specialization of attack tree A, and that attack tree B is a specialization of attack tree A from Fig. 2.

Example 1. First, consider the following assignment:

$a :=$ `"bribe sysadmin"` $\quad b :=$ `"break into system"`
$c :=$ `"install keylogger"` $\quad d :=$ `"steal backup"`

Then we have the following interpretations:

$$\begin{aligned}[\![A]\!] &= [\![\mathsf{AND}(\mathsf{OR}(a,d),\mathsf{SEQ}(b,c))]\!] \\ &= (a \sqcup_I d) \odot_I (b \rhd_I c)\end{aligned} \qquad \begin{aligned}[\![B]\!] &= [\![\mathsf{SEQ}(b,\mathsf{AND}(c,d))]\!] \\ &= b \rhd_I (c \odot_I d)\end{aligned}$$

$$\begin{aligned}[\![C]\!] &= [\![\mathsf{OR}(\mathsf{AND}(a,\mathsf{SEQ}(b,c)),\mathsf{SEQ}(b,\mathsf{AND}(c,d)))]\!] \\ &= (a \odot_I (b \rhd_I c)) \sqcup_I (b \rhd_I (c \odot_I d))\end{aligned}$$

We reuse the same names for base attacks across the interpretations above. Finally, we have the following two entailments:

$\underline{[\![C]\!] \leq_4 [\![A]\!]}$:	$\underline{[\![B]\!] \leq_I [\![A]\!]}$:
$(a \odot_I (b \rhd_I c)) \sqcup_I (b \rhd_I (c \odot_I d))$	$b \rhd_I (c \odot_I d)$
$\leq_4 (a \odot_I (b \rhd_I c)) \sqcup_I (b \rhd_I (d \odot_I c))$	$\leq_4 b \rhd_I (c \odot_I (a \sqcup_I d))$
$\leq_4 (a \odot_I (b \rhd_I c)) \sqcup_I (d \odot_I (b \rhd_I c))$	$\leq_4 b \rhd_I ((a \sqcup_I d) \odot_I c)$
$\leq_4 (a \sqcup_I d) \odot_I (b \rhd_I c)$	$\leq_4 (a \sqcup_I d) \odot_I (b \rhd_I c)$

Notice that neither $[\![A]\!] \leq_4 [\![C]\!]$ nor $[\![A]\!] \leq_4 [\![B]\!]$ hold, and thus, equivalences cannot prove the previous properties.

4 Lina: An EDSL for Conducting Threat Analysis using Causal Attack Trees

All of the models mentioned in this paper have been incorporated into a new embedded domain specific language (EDSL) for conducting threat analysis called Lina[6] which means small, young palm tree, but we constructed the name by combining the words linear and attack.

Lina is embedded inside of Haskell, a statically-typed functional programming language. The most important property of any EDSL is that they subsume the entirety of their host language, and can be prototyped quite rapidly. Haskell contributes several advantages, such as cutting edge verification tools, and a strong type system for catching bugs quickly.

Lina currently supports three types of causal attack trees:

- Process Attack Trees: these are attack trees with no attributes at all,
- Attributed Process Attack Trees: these are attack trees with attributes on the base attacks only. This is an intermediate representation used to build full attack trees.
- Full Attack Trees: these are attributed process attack trees with an associated attribute domain.

Internally, we represent causal attack trees by a simple data type, called `IAT`, whose nodes are labeled with an integer identifier we call `ID`. We then define each type of attack tree as a record (labeled tuple):

[6] Lina is under active development and its implementation can be found online at `https://github.com/MonoidalAttackTrees/Lina`

```haskell
-- Attributed Process Attack Tree
data APAttackTree attribute label =
 APAttackTree {
  process_tree :: IAT,
  labels :: B.Bimap label ID,
  attributes :: M.Map ID attribute
 }
```

```haskell
-- Process Attack Tree
type PAttackTree label = APAttackTree () label

-- Full Attack Tree
data AttackTree attribute label = AttackTree {
      ap_tree :: APAttackTree attribute label,
      configuration :: Conf attribute
}
```

A `B.Bimap` is a dictionary where we can efficiently look up `ID`s given a `label` or efficiently look up `label`s given an `ID`. A `M.Map` is a typical dictionary, and `()` is the unit type.

This design has several benefits. Internal attack trees are very easy to translate to various backends, especially formulas because we can use the `ID`s on base attacks as atomic formulas – which has its own benefits discussed below – and modifying labels and attributes is more efficient than having them labeled on the trees themselves. The previous data types reveal that actually all attack trees are attributed process attack trees, and a process attack tree simply does not use the attributes. This allows Lina to offer a uniform syntax for specifying all types of attack tree.

One important aspect of the definition of the various forms of attack trees is that the types `label` and `attribute` are actually type variables, and thus, our definition of attack trees is very general; in fact, `label` and `attribute` can be instantiated with any type whose elements are comparable. This property is captured by ad-hoc polymorphism using type classes in Haskell, and is checked during type checking.

Conducting threat analysis using attack trees requires them to be associated with an attribute domain. Typically, an attribute domain is a set, together with operations for computing the attribute of the branching nodes of an attack tree given attributes on the base attacks. In Lina attribute domains are defined by a type, here called `attribute`, and a configuration:

```haskell
data Conf attribute = (Ord attribute) => Conf {
  orOp  :: attribute -> attribute -> attribute,
  andOp :: attribute -> attribute -> attribute,
  seqOp :: attribute -> attribute -> attribute
}
```

Utilizing higher-order functions we can define configurations easily and generically. For example, here is the configuration that computes the minimum attribute for choice nodes, the maximum attribute for parallel nodes, and takes the sum of the children nodes as the attribute for sequential nodes:

```haskell
minMaxAddConf :: (Ord attribute,Semiring attribute) => Conf attribute
minMaxAddConf = Conf min max (.+.)
```

Notice here that this configuration will work with any type at all whose elements are comparable and form a semiring, thus making configurations generic and reusable. This includes types like `Integer` and `Double`.

The definitional language for attributed process attack trees of type `APAttackTree attribute label` is described by the following grammar:

$$at ::= \texttt{base\_na label} \mid \texttt{base\_wa attribute label} \mid \texttt{or\_node label at1 at2} \mid \texttt{and\_node label at1 at2} \mid \texttt{seq\_node label at1 at2}$$

A full example of the definition of an attributed process attack tree for attacking an autonomous vehicle can be found in Fig. 3. The definition of `vehicle_attack`

```
import Lina.AttackTree

vehicle_attack :: APAttackTree Double String
vehicle_attack = start_PAT $
  or_node "Autonomous Vehicle Attack"
    (seq_node "External Sensor Attack"
       (base_wa 0.2 "Modify Street Signs to Cause Wreck")
       (and_node "Social Engineering Attack"
          (base_wa 0.6 "Pose as Mechanic")
          (base_wa 0.1 "Install Malware")))
    (seq_node "Over Night Attack"
       (base_wa 0.05 "Find Address where Car is Stored")
       (seq_node "Compromise Vehicle"
          (or_node "Break In"
             (base_wa 0.8 "Break Window")
             (base_wa 0.5 "Disable Door Alarm/Locks"))
          (base_wa 0.1 "Install Malware")))
```

Fig. 3. Lina Script for an Autonomous Vehicle Attack.

begins with a call to `start_PAT`. Behind the scenes, all of the `ID`'s within the internal attack tree are managed implicitly, which requires the internals of Lina to work within a special state-based type. The function `start_PAT` initializes this state.

Finally, we can define the vehicle attack tree as follows:

```
vehicle_AT :: AttackTree Double String
vehicle_AT = AttackTree vehicle_attack minMaxMaxConf
```

This attack tree associates the vehicle attack attributed process attack tree with a configuration called `minMaxMaxConf` that simply takes the minimum as the attribute of choice nodes, and the maximum as the attribute of every parallel and sequential node.

Lina as two important features that other tools lack. First, it can abstract the definitions of attack trees. Second, it is highly compositional, because it is embedded inside of a functional programming language. Consider the following abstraction of `vehicle_attack`:

```
vehicle_AT' :: Conf Double -> AttackTree Double String
vehicle_AT' conf = AttackTree vehicle_attack conf
```

Here the configuration has been abstracted. This facilitates experimentation because the security practitioner can run several different forms of analysis on the same attack tree using different attribute domains.

Attack trees in Lina can also be composed and decomposed; hence, complex trees can be broken down into smaller ones, then studied in isolation. This helps

```
se_attack :: APAttackTree Double String
se_attack = start_PAT $
  and_node "social engineering attack"
     (base_wa 0.6 "pose as mechanic")
     (base_wa 0.1 "install malware")
```

```
bi_attack :: APAttackTree Double String
bi_attack = start_PAT $
  or_node "break in"
     (base_wa 0.8 "break window")
     (base_wa 0.5 "disable door alarm/locks")
```

```
cv_attack :: APAttackTree Double String
cv_attack = start_PAT $
  seq_node "compromise vehicle"
    (insert bi_attack)
    (base_wa 0.1 "install malware")
```

```
es_attack :: APAttackTree Double String
es_attack = start_PAT $
  seq_node "external sensor attack"
       (base_wa 0.2 "modify street signs to cause
                     wreck")
       (insert se_attack)
```

```
on_attack :: APAttackTree Double String
on_attack = start_PAT $
  seq_node "overnight attack"
     (base_wa 0.05 "Find address where car
                    is stored")
     (insert cv_attack)
```

```
vehicle_attack'' :: APAttackTree Double String
vehicle_attack'' = start_PAT $
  or_node "Autonomous Vehicle Attack"
    (insert es_attack)
    (insert on_attack)
```

Fig. 4. The Autonomous Vehicle Attack Decomposed

facilitate correctness, and offers more flexibility. As an example, in Fig. 4 we break up `vehicle_attack` into several smaller attack trees. We can see in the example that if we wish to use an already defined attack tree in an attack tree we are defining, then we can make use of the `insert` function. As we mentioned above, behind the scenes Lina maintains a special state that tracks the identifiers of each node; thus, when one wishes to insert an existing attack tree, which will have its own identifier labeling, into a new tree, then that internal state must be updated; thus, `insert` carries out this updating. Lina is designed so that the user never has to encounter that internal state.

So far we have introduced Lina's basic design and definitional language for specifying causal attack trees, and we have already begun seeing improvements over existing tools; however, Lina has so much more to offer. We now introduce Lina's support for reasoning about and performing analysis on causal attack trees.

Kordy et al. [8] introduce the SPTool, an equivalence checker for causal attack trees that makes use of the rewriting logic system Maude [3] which allows one to specify rewrite systems and systems of equivalences. Kordy et al. specify the equivalences for causal attack trees from Jhawar et al.'s [6] work in Maude, and then use Maude's querying system to automatically prove equivalences between causal attack trees. This is a great idea, and we incorporate it into Lina, but we make several advancements over SPTool.

Lina includes a general Maude interface, and allows the user to easily define new Maude backends, where a *Maude backend* corresponds to a Maude specification of a particular rewrite system. Currently, Lina has two Maude backends: equivalences for causal attack trees, and the multiplicative attack tree linear logic (MATLL). The former is essentially the exact same specification as the SPTool, but the latter corresponds to the quaternary semantics defined in Section 3 and

Section **??**; specifically, this backend is defined as a rewrite system that includes all of the rules from Lemma **??** and Lemma **??**.

Attributed process attack trees are converted into the following syntax:

$$\text{(Maude Formula) } F := \texttt{ID} \mid F1; F2 \mid F1.F2 \mid F1 + F2$$

This is done by simply converting the internal attack tree into the above syntactic form. For example, the Maude formula for the autonomous vehicle attack from Fig. 3 is `(0 ; (1 . 2)) || (5 ; ((6 || 7) ; 2))`, where each integer corresponds to the identifier of the base attacks. Note that the base attack `2` appears twice, this is because this base attack appears twice in the original attack tree. This syntax is then used to write the Maude specification for the various backends.

The full Maude specification for the causal attack tree equivalence checker can be found in Appendix A. However, Kordey et al.'s specification only supports proving equivalences, but what about specializations? Lina supports proving specializations between attack trees using the MATLL Maude backend. Its full Maude specification can be found in Fig. 5. The axioms `a1` through `a5` are

```
mod MATLL is

protecting LOOP-MODE .

sorts Formula .
subsort Nat < Formula .

op _||_  : Formula Formula -> Formula [ctor assoc comm] .
op _._   : Formula Formula -> Formula [ctor assoc comm prec 41] .
op _;_   : Formula Formula -> Formula [ctor assoc prec 40] .

var a b c d : Formula .

rl [a1]          : a . (b || c)       => (a . b) || (a . c) .
rl [a1Inv]       : (a . b) || (a . c) => a . (b || c) .
rl [a2]          : a ; (b || c)       => (a ; b) || (a ; c) .
rl [a2Inv]       : (a ; b) || (a ; c) => a ; (b || c) .
rl [a3]          : (b || c) ; a       => (b ; a) || (c ; a) .
rl [a3Inv]       : (b ; a) || (c ; a) => (b || c) ; a .
rl [a4]          : (a . b) ; c        => a . (b ; c) .
rl [a4Inv]       : a . (b ; c)        => (a . b) ; c .
rl [a5]          : (a ; b) . (c ; d)  => (a . c) ; (b . d) .
rl [a5Inv]       : (a . c) ; (b . d)  => (a ; b) . (c ; d) .
rl [switch]      : a ; (b . c)        => b . (a ; c) .
rl [seq-to-para] : a ; b              => a . b .
endm
```

Fig. 5. Maude Specification for MATLL.

actually equivalences, but the last two rules are not. At this point we can use these backends to reason about attack trees.

The programmer can make queries to Lina by first importing one or more Lina modules, and then making a query using Haskell's REPL – read, evaluate,

print, loop – called GHCi. Consider the example Lina program in Fig. 6. These are the attack trees from Fig. 2. Then an example Lina session is as follows:

```
import Lina.AttackTree
import Lina.Maude.MATLL

-- A
enc_data1 :: PAttackTree String
enc_data1 = start_PAT $
  and_node "obtain secret"
    (or_node "obtain encryped file"
       (base_na "bribe sysadmin")
       (base_na "steal backup"))
    (seq_node "obtain password"
       (base_na "break into system")
       (base_na "install keylogger"))

-- C
enc_data2 :: PAttackTree String
enc_data2 = start_PAT $
  or_node "obtain secret"
    (and_node "obtain secret via sysadmin"
       (base_na "bribe sysadmin")
          (seq_node "obtain password"
             (base_na "break into system")
             (base_na "install keylogger")))
    (seq_node "break in, then obtain secrect"
       (base_na "break into system")
       (and_node "obtain secret from inside"
         (base_na "install keylogger")
         (base_na "steal backup")))
```

Fig. 6. Full Lina Script for the Attack Trees A and C from Fig. 2.

```
> :load source/Lina/Examples/Specializations.hs
...
Ok, modules loaded
> is_specialization enc_data2 enc_data1
True
>
```

In this session we first load the Lina script from Fig. 6 which is stored in the file `Specializations.hs`. Then we ask Lina if `enc_data2` is a specialization of `enc_data1`, and Lina responds `True`, thus automating the proof given in Example 1.

In addition to reasoning about attack trees, Lina also support analysis of attack trees. Currently, Lina supports several types of analysis: evaluating attack trees, querying the attack tree for the attribute value of a node, projecting out the set of attacks from an attack tree, and computing the maximal and minimal attack.

When one defines an attack tree that tree is left unevaluated; that is, the attribute dictionary associated with the attack tree only has attributes recorded for the base attacks. If one wishes to know the attribute values at the branching

nodes, then one must evaluate the attack tree, which populates the attribute dictionary with the missing attributes. For example, we may evaluate the attack tree for the autonomous vehicle attack from Fig. 3, and query the tree for the attributes at various nodes:

```
> let (Right e_vat) = eval vehicle_AT
> e_vat <@> "social engineering attack"
0.6
>
```

Here we first evaluate the attack tree `vehicle_AT` giving it the name `e_vat`, and then we use the attributed query combinator `<@>` to ask for the attribute at the parallel node labeled with `"social engineering attack"`. Note that the evaluator, `eval`, uses the configuration associated with the attack tree to compute the values at each branching node.

It is also possible to project out various attacks from an attack tree. In Lina an *attack* corresponds to essentially an attack tree with no choice nodes. We call its data type `Attack attribute label`. An attack does not have any choice nodes, because they are all split into multiple attacks; one for each child node. For example, the set of possible attacks for the autonomous vehicle attack from Fig. 3 can be found in Fig. 7. Lina can compute these automatically using the

```
SEQ("external sensor attack",0.6)
        ("modify street signs to cause wreck",0.2)
        (AND("social engineering attack",0.6)
                ("pose as mechanic",0.6)
                ("install malware",0.1))
```

```
SEQ("over night attack",0.8)
        ("Find address where car is stored",0.05)
        (SEQ("compromise vehicle",0.8)
                ("break window",0.8)
                ("install malware",0.1))
```

```
SEQ("over night attack",0.5)
        ("Find address where car is stored",0.05)
        (SEQ("compromise vehicle",0.5)
                ("disable door alarm/locks",0.5)
                ("install malware",0.1))
```

Fig. 7. Set of Possible Attacks for an Autonomous Vehicle Attack.

`get_attacks` command. Finally, given the set of attacks for the autonomous vehicle attack we can also compute the set of minimal and maximal attacks. For example, consider the following session:

```
> min_attacks.get_attacks $ vehicle_AT
[SEQ("over night attack",0.5)
("Find address where car is stored",0.05)
(SEQ("compromise vehicle",0.5)
```

```
("disable door alarm/locks",0.5)
("install malware",0.1))]
```

In this session we first apply `get_attacks` to `vehicle_AT` to compute the set of possible attacks, and then we compute the minimal attack from this set.

5 Conclusion and Future Work

We made two main contributions: a new four-valued truth table semantics of causal attack trees that supports specializations of attack trees, and a new embedded domain specific programming language called Lina for specifying, reasoning, and analyzing attack trees.

We plan to lift the quaternary semantics into a natural deduction system based on the logic of bunched implications, and then study proof search within this new system. Lina is under active development, and we have a number of extensions planned, for example, adding support for attack-defense trees, attack(-defense) graphs, attack nets, a GUI for viewing the various models, and a SMT backend. Finally, it is necessary for number of case studies to be carried out within Lina to be able to support the types of analysis required for real world applications.

6 Acknowledgments

This work was supported by NSF award #1565557. We thank Clément Aubert for helpful discussions and feedback on previous drafts of this paper.

References

1. S.A. Camtepe and B. Yener. Modeling and detection of complex attacks. In *Security and Privacy in Communications Networks*, pages 234–243, Sept 2007.
2. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.
3. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.1). *SRI International, Menlo Park*, 2005.
4. Olga Gadyatskaya and Rolando Trujillo-Rasua. New directions in attack tree research: Catching up with industrial needs. In Peng Liu, Sjouke Mauw, and Ketil Stolen, editors, *Graphical Models for Security*, pages 115–126, Cham, 2018. Springer International Publishing.
5. Ross Horne, Sjouke Mauw, and Alwen Tiu. Semantics for specialising attack trees based on linear logic. *Fundamenta Informaticae*, 153(1-2):57–86, 2017.
6. Ravi Jhawar, Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Rolando Trujillo-Rasua. Attack trees with sequential conjunction. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 339–353. Springer International Publishing, 2015.

7. Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
8. Barbara Kordy, Piotr Kordy, and Yoann van den Boom. *SPTool – Equivalence Checker for* `SAND` *Attack Trees*, pages 105–113. Springer International Publishing, Cham, 2017.
9. Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Foundations of attack–defense trees. In Pierpaolo Degano, Sandro Etalle, and Joshua Guttman, editors, *Formal Aspects of Security and Trust*, pages 80–95, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
10. Barbara Kordy, Marc Pouly, and Patrick Schweitzer. Computational aspects of attack–defense trees. In Pascal Bouvry, MieczysławA. Kłopotek, Franck Leprévost, Małgorzata Marciniak, Agnieszka Mykowiecka, and Henryk Rybiński, editors, *Security and Intelligent Information Systems*, volume 7053 of *Lecture Notes in Computer Science*, pages 103–116. Springer Berlin Heidelberg, 2012.
11. Barbara Kordy, Marc Pouly, and Patrick Schweitzer. A probabilistic framework for security scenarios with dependent actions. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods*, volume 8739 of *Lecture Notes in Computer Science*, pages 256–271. Springer International Publishing, 2014.
12. Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In DongHo Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer Berlin Heidelberg, 2006.
13. J. P. McDermott. Attack net penetration testing. In *Proceedings of the 2000 Workshop on New Security Paradigms*, NSPW '00, pages 15–21, New York, NY, USA, 2000. ACM.
14. Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th international workshop on types in language design and implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.
15. L. Piètre-Cambacédès and M. Bouissou. Beyond attack trees: Dynamic security modeling with boolean logic driven markov processes (bdmp). In *Dependable Computing Conference (EDCC), 2010 European*, pages 199–208, April 2010.
16. Bruce Schneier. Attack trees: Modeling security threats. *Dr. Dobb's journal*, December 1999.
17. Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, August 2014.

A Maude Specification for Causal Attack Trees

```
mod Causal is

protecting LOOP-MODE .

sorts Formula .
subsort Nat < Formula .

op _||_    : Formula Formula -> Formula [ctor assoc comm] .
op _._     : Formula Formula -> Formula [ctor assoc comm] .
op _;_     : Formula Formula -> Formula [ctor assoc] .
```

```
op EQ(_,_) : Formula Formula -> Bool .

var P Q R S : Formula .

eq P . (Q || R)  = (P . Q)  || (P . R) .
eq P ; (Q || R)  = (P ; Q)  || (P ; R) .
eq (Q || R) ; P  = (Q ; P)  || (R ; P) .

ceq EQ(P,Q) = true
  if P = Q .
eq EQ(P,Q) = false .

endm
```